AFRL-IF-RS-TR-2005-21
Interim Technical Report
February 2005

ADVICE AND LEARNING IN PROBLEM SOLVING

Kestrel Institute

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K506

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-21 has been reviewed and is approved for publication

APPROVED: /s/

JAMES M. NAGY
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Acting Chief
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	FEBRUARY 2005	Interim Aug 03 – Nov 04

4. TITLE AND SUBTITLE
ADVICE AND LEARNING IN PROBLEM SOLVING

5. FUNDING NUMBERS
C - F30602-00-C-0209
PE - 62301E
PR - DASA
TA - 00
WU - 05

6. AUTHOR(S)
Richard Wadinger, Marcel Becker, Alessandro Coglio,
David Cyrluk, Stephen Fitzpatrick, Limei Gilham,
Cordell Green, Douglas Smith, and Stephen Westfold

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kestrel Institute
3260 Hillview Avenue
Palo Alto California 94304

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency AFRL/IFT
3701 North Fairfax Drive 525 Brooks Road
Arlington Virginia 22203-1714 Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-21

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: James M. Nagy/IFT/(315) 330-3173/ James.Nagy@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Respect is an effort to allow a problem solver to accept advice and exhibit learning. It is based on a high-level description of a problem solver that can be examined and manipulated by the problem-solver itself. Experiments are being conducted in such problem-solving arenas as meeting scheduling, type checking, and logistics planning. By specializing a general-purpose problem solver to a declarative problem specification, we obtain a procedure for that particular problem domain, which is not necessarily efficient. Because the initial problem solver is high-level, the information is explicitly available as to which operations can be reordered; reordering can lead to radical changes in the search space. In the meeting scheduling domain, this reordering can ensure that we schedule the scarce resources first (busy people, popular facilities). We have seen examples in which failed solutions can be examined automatically to suggest relaxation of constraints---a kind of problem reformulation. For example, if a scheduling problem is unsolvable, an examination of the search space can suggest that a trip be extended an additional day, or that some participants work an extra hour. The envisioned system accepts advice as to how to examine the search space.

14. SUBJECT TERMS
Problem Solving, Learning, Advice, Specware, Theorem Proving, SNARK, Meta-Level Reasoning, Meeting Scheduling, Type Inference

15. NUMBER OF PAGES
53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

List of Figures

Summary

Respect is an effort to allow a problem solver to accept advice and exhibit learning. It is based on a high-level description of a problem solver that can be examined and manipulated by the problem-solver itself. Experiments are being conducted in such problem-solving arenas as meeting scheduling, type checking, and logistics planning.

By specializing a general-purpose problem solver to a declarative problem specification, we obtain a procedure for that particular problem domain, which is not necessarily efficient. Because the initial problem solver is high-level, the information is explicitly available as to which operations can be reordered; reordering can lead to radical changes in the search space. In the meeting scheduling domain, this reordering can ensure that we schedule the scarce resources first (busy people, popular facilities).
We have seen examples in which failed solutions can be examined automatically to suggest relaxation of constraints--a kind of problem reformulation. For example, if a scheduling problem is unsolvable, an examination of the search space can suggest that a trip be extended an additional day, or that some participants work an extra hour. The envisioned system accepts advice as to how to examine the search space.

Introduction

Current problem-solving systems, however competent they may be, are rigid in that they cannot improve their performance through experience and they cannot accept advice from a knowledgeable consultant. One reason for this is that a problem-solving system does not have access to the design decisions of its implementer, and hence is not in a position to alter them. In the Respect project, we envision a problem solver that has access and control over its entire design history, and hence is in a position to alter itself as a result of advice or experience. On selected examples, we have shown how the derivation structure identifies choice points at which advice is called for, and shows how the appropriate advice can effect drastic improvements in the search space and the quality of the final product. Creating and retaining a record of the derivation structure also makes us better able to adapt to changing task descriptions and environments—we can develop versions of the problem solver that are specialized to the particular task at hand.

Advice might be abstract and domain-independent (work on shorter goals first) or domain-specific (schedule busier people first; assign most constrained types first). Advice can express assistance in problem solving (delete goals that are subsumed by others.) or preferences between solutions (prefer to have meetings end early; prefer shorter business trips; prefer to use less fuel.)

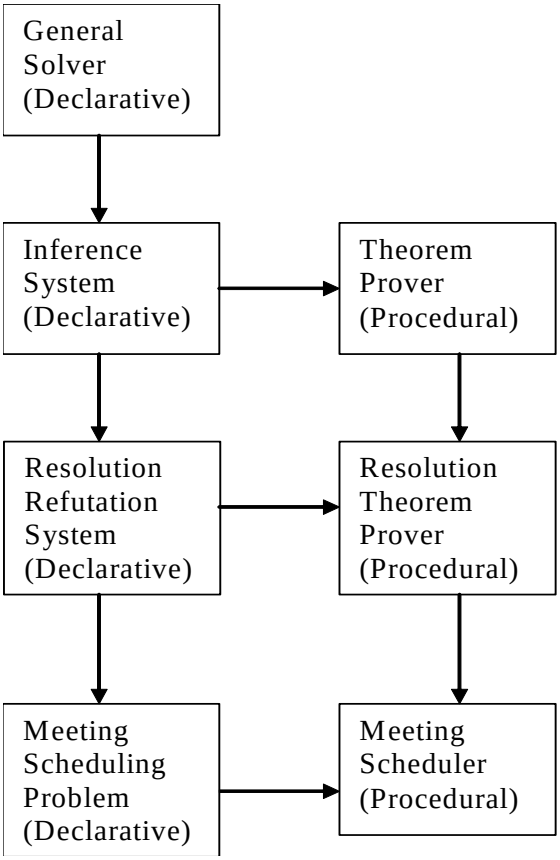

Figure 1: The Derivation Structure

We begin (see Figure 1) with a specification of a problem solver and a derivation within Planware of a general-purpose problem solver that meets that specification. We then apply the problem solver to a particular subject domain theory. The result is a problem solver specialized to the domain. There is no expectation at this point that this problem solver will be efficient. The general problem solver has been implemented without any knowledge of the intended application.

At each stage of the derivation, the system or implementer has made choices. Having access to those choices gives us the possibility of changing them. In general, a choice corresponds to choosing an element from a set. We may choose an implementation for a

data structure from a variety of possible implementations; we may choose the place to insert a task in an agenda of tasks; we may choose a subformula from the set of subformulas of a given formula. Each choice represents a decision that can be altered, giving possibly radical alterations of the search space. The choice points indicate places at which a system can ask for advice from a knowledgeable consultant.

There is a virtue in having access to the many levels in the derivation of the problem solver. For instance, while the highest level may be declarative, advice about what to do first is often at a procedural level, in which actions are taken. Domain-independent advice will be at an abstract level, while domain-specific advice will necessarily be at a concrete level.

While there are many possible problem solvers, in our experiments we have looked at systems for using logical inference as a problem solver. The methods are entirely general. Indeed, the design of Respect allows us to alter that choice and examine other problem-solving structures when they are more appropriate for a particular subject domain. Our intention is to incorporate these ideas into a version of the software-development environment Specware [Kestrel].

The specifications on the left in Figure 1 are declarative—they indicate what is to be done without indicating how. The other specifications are procedural—they indicate what actions are to be performed, but do not say why. The entire derivation structure expresses both the why and the how.

The higher-level specifications are more abstract, while the lower-level ones are more concrete. While the top specification just talks about what it means to solve a problem, the second level talks about theorem proving, the third level to resolution theorem proving, and the lowest level is specific to a particular application, meeting scheduling.

While not all the links in the structure have been implemented, we have done enough to suggest the value of exploring this approach further. We have shown that choice points in the structure indicate where advice can be accepted and that advice can have a dramatic difference in both the quality of the solution and the time spent in finding it. We will focus on particular aspects of the derivation structure.

Detailed Discussion of the Derivation Structure: Inference

In this section we spell out some of the details of some of the specifications involved in the derivation of a planning system.

In Appendix 1: Generic Declarative Refutation Theorem Prover we see the specification for an inference system that is independent of the particular inference rules to be applied; to implement this system, we must select an inference rule. Hence advice such as "use resolution" or "use rewriting" would be meaningful. On the other hand, because it is declarative, it would not be meaning to give the advice "treat short sentences before long sentences" at this level–that is procedural advice.

In Appendix 2: Declarative Resolution Theorem Prover, the inference rule has been specialized to binary resolution. For these experiments this was done by hand, but ultimately it would be carried out within Planware. Also, while the present specification is for propositional logic, it could be converted to first-order logic by the introduction of substitution arguments and invocation of unification.

While the previous theorem prover was declarative, the next (Appendix 3: Procedural Resolution Theorem Prover) is an actual MetaSlang procedure. At this level, we can express strategic advice, such as stating which sentences to apply the resolution rule to first. Because this level is still domain-independent, we cannot yet specify domain-specific advice. In the next section, we see how knowing the subject domain allows us to express such advice.

Meeting-Scheduling Theories

To experiment with the impact of advice and learning on particular subject domains, we have developed declarative and procedural versions of particular domain theories, including meeting scheduling and type inference. While ultimately we will develop a single theory that will allow us to specify a variety of scheduling problems, for the purpose of these experiments we have developed theories to specify particular problems. While ultimately we will use the theorem prover derived in the previous section, to start off the bootstrap process we have used the theorem prover SNARK that is built into Specware. Scheduling has been one of the more successful applications of the Specware family of problem solvers [Smith et al.]

In Appendix 4: Declarative Specification of a Simple Meeting-Scheduling Problem, we specify a simple meeting-scheduling problem. This was an early experiment in different representations of the domain. In this problem, there are three participants, Art, Bob, and Carol, who are busy at three times, A, B, and C, respectively. While in later problems we deal with real time intervals, at this stage we regard times as abstract. To say that Art is free at Time B, we say

```
Art(TimeB),
```

and so forth.

To schedule a meeting between Bob and Carol, we say

```
ex(time1 : Time)
   Bob(time1) & Carol(time1).
```

In other words, we prove the existence of a time `time1` at which both Bob and Carol are free. During the proof, the variable `time1` will be replaced by a concrete time `TimeA`, which represents the only time at which Bob and Carol are free. This solution is extracted from the proof by the theorem prover. Solution was essentially instantaneous.

While declarative specifications resemble logic programs superficially, in fact they are quite different. A logic program represents a procedure in logical notation. A declarative specification is a simple statement of facts with no order of execution or other procedural information implied.

In Appendix 5: Procedural Meeting-Scheduling Theory, we give a MetaSlang program that represents a meeting-scheduling procedure. Eventually this would be developed by specializing the procedural version of a problem solver, such as the binary resolution theorem prover, to a declarative statement of the meeting-scheduling theory.

In Appendix 6: Two-Meeting Scheduling Problem, we deal with a problem in which two meetings had to be scheduled and one of the participants was more busy than the other two. The intention was to show that the advice to schedule the busiest participant first would lead to more efficient scheduling. However, the solution of the problem, with or without advice, was also instantaneous.

In Appendix 7: Scheduling with Space, we introduce space as well as time–to schedule a meeting, one must find a room as well as a time. This requires a change in representation. We introduce two predicate symbols, `FreePerson` and `FreeRoom`. We deal with two distinct times, `Morning` and `Afternoon`, and two rooms, `LightRoom` and `DarkRoom`. To state that Carol is free in the afternoon, we write

```
FreePerson(Carol, Afternoon).
```

To state that the light room is free in the morning, we say

```
FreeRoom(LightRoom, Morning).
```

To state a problem of scheduling two meetings, we say

```
ex(time1 : Time, time2 : Time, room1 : Room, room2 : Room,
     schedule : Time * Room * Time * Room)
   FreePerson(Art, time1) & FreePerson(Bob, time1) &
   FreeRoom(room1, time1) &
   FreePerson(Bob, time2) & FreePerson(Carol, time2) &
   FreeRoom(room2, time2) &
   ~(time1 = time2) &
   schedule = (time1, room1, time2, room2).
```

In other words, we want to find a time `time1` at which Art, Bob, and a room `room1` are all free, and another time `time2` at which Bob, Carol and a room `room2` are all free; if we succeed, a successful schedule is to have the first meeting be at `time1` in `room1` and the second meeting be at `time2` in `room2`.

All of the above problems were too simple to be useful in experimenting with the effects of advice. In the next problem (Appendix 8: Advice about Constraint Ordering), we expanded the number of participants and rooms, in an attempt to find a problem that would benefit from advice. We posited that there are nine participants, eight times, and eleven rooms. While most people and rooms are free at all times, Bob and the dark room are only free at Time 5. We are required simply to find a time and room at which all nine are available.

Because Bob is the busiest person and the dark room is the busiest room, our expectation was that the theorem prover would find a solution more quickly if it was given the advice to schedule these most constrained resources first. We say this by specifying, in effect that Bob » Art, that DarkRoom » LightRoom, etc. We experimented with giving this

advice, giving a perversely bad ordering in which the most constrained resources were scheduled last, and using no ordering strategy at all.

Somewhat to our surprise, the theorem prover behaved equally well with the good advice and with the bad advice, and slightly worse without an ordering strategy.

The explanation was that giving any ordering eliminates redundancy, because it doesn't attempt to handle the same set of constraints in different orders. General-purpose theorem-proving strategies, which are built into the theorem prover, made the domain-specific advice we used redundant. The most important of these strategies here is to favor shorter clauses over longer ones. In particular, unit clauses, such as the descriptions of the free times of people and rooms, are handled immediately. For the meeting-scheduling problem, the clause in which Bob is scheduled to meet at Time 5 is shorter than the clauses in which Bob is scheduled to meet at other times, since the constraint that he be free at Time 5 is handled immediately by resolution with a unit clause.

While in the current implementation the advice to do short clauses first is built in, when we derive a theorem prover from an abstract specification we expect to be able to accept advice that suggests handling shorter clauses first.

In our previous examples, ordering advice told the problem solver what to attempt first. But there is another kind of ordering advice, which expresses which kinds of solutions are to be preferred. This might better be regarded as part of the specification, but it does not constitute a hard restriction on the solution; rather, it states that if two solutions are possible, one of them might be preferred to the other. For this reason, such preferences are sometimes known as *soft constraints.*

We have experimented with a domain-independent way of dealing with such problems. We first ignore the preference advice and find a single solution to the problem. We then search for additional solution, adding to the specification the condition that the new solution be "better" than the original, where "better" means preferable in the sense expressed by the advice. If we succeed, again, we seek a third solution that is "better" than the second, and so on until either the set of solutions is exhausted or we exceed some preset time limit.

In Appendix 9: Preference Advice, we consider a problem in which four different meetings are to be scheduled, some but not all of which can be concurrent. While it is perfectly permissible to have them scattered throughout the day, we state a preference that all the meetings should be over as early as possible. With this preference advice, and the above-mentioned method of invoking a theorem prover repeatedly to seek better and better solutions, we find a solution in which the meetings are held as early as possible, and with the greatest possible overlap.

While in the previous problem we dealt with advice about a preference in time (finishing earlier is better), in the next problem (in Appendix 10: Opportunistic Meeting Scheduling), we considered advice about a preference in distance.

We suppose that a meeting must be scheduled between Alice and Bob during the first ten days of April. (This is the first problem in which we deal with concrete time intervals.) Alice lives in San Francisco and Bob lives in New York. One solution is to have Alice make a special trip to New York, Bob make a special trip to San Francisco, or both of them make a special trip to a third location. However, we are told that Alice already has a trip scheduled to Minneapolis, Minnesota, April 3–7, and Bob has a trip scheduled to Saint Paul, Minnesota, April 5–9. These are overlapping time intervals; the theorem prover used a built-in temporal reasoning procedure to detect this. Also, Minneapolis and Saint Paul are within six miles of each other; to detect this, the theorem prover invokes an external gazetteer (for finding their latitudes and longitudes) and an geographical computation procedure (for finding the distance between those latitude/longitude pairs). Consequently it proposes a better solution in which they meet in Minnesota during April 5–7.

Whereas learning is sometimes associated with inductive inference or statistics, in Appendix 11: Learning in Meeting Scheduling, we treat learning as a way of responding to a failed solution attempt to exhibit better performance in the future, in this case by negotiating a restatement of the problem.

We start with an impossible problem, in which three distinct meetings must be scheduled in two time-slots. Art, Bob, and Carol are free at Times 1 and 2 but busy at Time 3. The theorem prover fails to find a solution.

However, if we examine the search space, we find that every branch ends with a two-literal clause that requests that two of the participants be free at Time 3, e.g,

```
FreePerson(Art, Time3) & FreePerson(Bob, Time3),
```
or
```
FreePerson(Bob, Time3) & FreePerson(Carol, Time3).
```

Such conditions can be detected automatically via the *abduction* facility of the theorem prover. This suggests that we relax the constraints of the problem so that two of the participants are free at a third time.

Type-Inference Theory

While most of our experiments have been conducted in the domain of meeting scheduling, it is intended that the techniques we explore be valuable across many domains. For example, we have found then to be useful for type inference, the consistent assignment of types to the symbols in an expression (see Appendix 12: Type-Inference Theory). For instance, it is appropriate to offer advice to assign a type to the most constrained symbol first. These results are of special interest because Specware itself uses type inference, so any improvements will be immediately applicable.

Now we reproduce the diagram of the Introduction, indicating which appendices, including content or advice, correspond to which components of the figure.

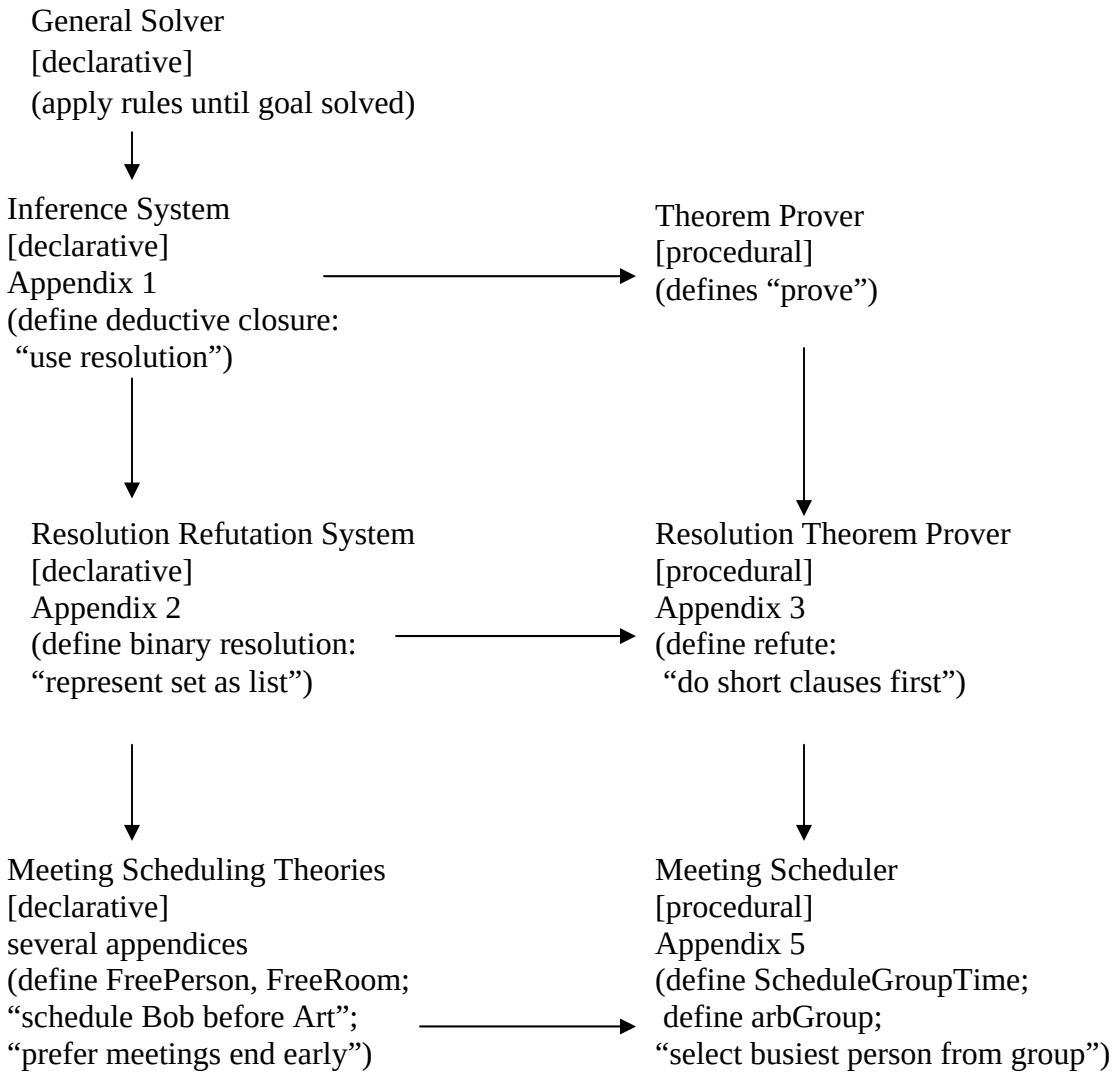

Figure 2: The Derivation Structure Revisited

Related Work

[Weyhrauch] developed a formal metatheory for first-order logic and introduced a "reflection principle," a rule of inference that allowed conclusions about logic derived by the metatheory to alter and improve the metatheory itself. His work was in the setting of an interactive theorem prover.

[Meseguer] develops a metalogical framework for describing other logics. Huet and Plotkin collect descriptions of logical frameworks. In this work, the emphasis is on representing logical reasoning, rather than proof search or automated problem solving.

[Boyer] use a reflection principle similar to Weyhrauch's in their automatic theorem prover, so that a theorem about arithmetic expressions proved by the system can accelerate the performance of the system itself when applied to arithmetic expressions.

[Apt] develops a theory of problem solvers in which resolution can be viewed as an instance of a constraint problem solver.

[Ridge] derives a first-order-logic theorem prover from a soundness and completeness proof for a system of first-order logic. This proof, and the extraction of the theorem prover, constitutes a full derivation history for the theorem prover. The theorem prover derived has no function symbols or equality and cannot extract answers from proofs, but these are plausible extensions.

Discussion

This work is still in its early stages but it suggests that having access to the derivation structure of a problem solver can allow us to effect radical improvements in its performance. The choice points in the refinement are natural places to ask advice from a consultant. Since the advice can alter the derivation, it can have a radical effect on the performance of the problem solver in the selected subject domain. Advice can be declarative (e.g., preferences over solutions) or procedural (e.g., do unit clauses first); abstract (do short clauses first) or domain-specific (schedule the most constrained people or resources first). Therefore it is crucial to have access to all levels of the derivation structure.

There is also the possibility of improving the problem solving structure itself by reasoning in problem-solving theories. Thus reasoning about type inference can result in improvements in the type inference we use in reasoning. Reasoning about unification or deduction can improve the unification and deduction mechanism. If we derive a unification algorithm for a theory that is used in our derivation process, we can use that algorithm to expedite future derivations.

Examination of the search space of sample problems can lead to alterations in the problem solver or even renegotiation of the specifications for problems and the relaxation of constraints. The ability to rerun the problem solver and revise the specification provides a novel way of dealing with user advice on preferences. These results apply across a variety of domains, including meeting scheduling, type inference, and logistics planning.

Appendix 1: Generic Declarative Refutation Theorem Prover

Here is the specification for a generic refutation theorem prover:

```
deduction_spec = spec

(* comment:
This is a declarative spec for a generic refutation theorem prover;
it computes the deductive closure of a set of sentences with respect
to a set of inference rules.  For a refutation procedure we check
that the false sentence occurs in the closure.
   end comment *)

sort Sentence

sort SentenceSet = {s : Sentence | true}

op inSentenceSet : Sentence * SentenceSet -> Boolean

op subSentenceSet : SentenceSet * SentenceSet -> Boolean

sort Rule

sort RuleSet = {r : Rule | true}

op inRuleSet : Rule * RuleSet -> Boolean

op subRuleSet : RuleSet * RuleSet -> Boolean

op  apply_rs : Rule * Sentence -> SentenceSet

op apply_Rs : RuleSet * Sentence -> SentenceSet

op apply_rS : Rule * SentenceSet -> SentenceSet

op apply_RS : RuleSet * SentenceSet -> SentenceSet

axiom definition_of_apply_RS is
 fa(R : RuleSet, S : SentenceSet, s : Sentence)
  (inSentenceSet(s, apply_RS(R, S)) <=>
   (ex(r : Rule)
    (inRuleSet(r, R) &
     inSentenceSet(s, apply_rS(r, S)))))

axiom monotonicity_of_RuleSet is
  fa(R1 : RuleSet, R2 : RuleSet, S : SentenceSet)
    subRuleSet(R1, R2) =>
    subSentenceSet(apply_RS(R1, S), apply_RS(R2, S))

axiom monotonicity_of_SentenceSet is
  fa(R : RuleSet, S1 : SentenceSet, S2 : SentenceSet)
    subSentenceSet(S1, S2) =>
```

```
    subSentenceSet(apply_RS(R, S1), apply_RS(R, S2))

op neg : Sentence -> Sentence

op implies : Sentence * Sentence -> Boolean

op conjunction : SentenceSet -> SentenceSet

op impliesSet : SentenceSet * SentenceSet -> Boolean

axiom definition_of_impliesSet is
  fa(S1 : SentenceSet, S2 : SentenceSet)
   (impliesSet(S1, S2) <=>
    (fa(s2 : Sentence)
      (inSentenceSet(s2, S2) =>
       (ex(S11 : SentenceSet)
        subSentenceSet(S11, S1) &
         implies(conjunction(S11), s2)))))

op closure : RuleSet * SentenceSet -> SentenceSet

axiom closure_is_implied is
 fa(R : RuleSet, S : SentenceSet)
  impliesSet(S, closure(R, S))

axiom closure_is_closed is
 fa(R : RuleSet, S : SentenceSet)
  apply_RS(R, closure(R, S)) = closure(R, S)

axiom closure_is_smallest is
 fa(R : RuleSet, S : SentenceSet, S1 : SentenceSet)
  (impliesSet(S, S1) &
   apply_RS(R, S1) = S1) =>
   subSentenceSet(closure(R, S), S1)

op prover : RuleSet * SentenceSet ->  Option Boolean

op true_sentence : Sentence
op false_sentence : Sentence
op RuleSet0 : RuleSet

axiom completeness_of_prover is
 fa (S : SentenceSet)
  (implies(conjunction(S), false_sentence) =>
  restrict   (prover(RuleSet0, S)) )

axiom soundness_of_prover is
 fa(S : SentenceSet)
  (some(prover(RuleSet0, S)) =>
   implies(conjunction(S), false_sentence))
endspec
```

Appendix 2: Declarative Resolution Theorem Prover

Here is the declarative specification of a propositional resolution theorem prover. Although developed manually, it will ultimately be developed as a refinement of the previous specification.

```
theorem_prover_spec = spec

sort Sentence

op true_sentence : Sentence
op false_sentence : Sentence
op neg : Sentence -> Sentence

sort SentenceSet = {s : Sentence | true}

%op rule? :  SentenceSet * Sentence -> Boolean

sort Rule = SentenceSet * Sentence -> Boolean

sort RuleSet = {r : Rule | true}

op apply_rS : Rule * SentenceSet * Sentence -> Boolean

op refute : SentenceSet * RuleSet -> Boolean

op add_S : Sentence * SentenceSet -> SentenceSet

axiom add_S_collapses is
 fa(s : Sentence, S : SentenceSet)
  add_S(s, add_S(s, S)) = add_S(s, S)

op empty_S : SentenceSet

op single_S : Sentence -> SentenceSet

op duo_S : Sentence * Sentence -> SentenceSet

axiom definition_of_single_S is
 fa(s : Sentence)
  single_S(s) = add_S(s, empty_S)

axiom definition_of_duo_S is
 fa(s1 : Sentence, s2 : Sentence)
  duo_S(s1, s2) = add_S(s1, single_S(s2))

op add_R : Rule * RuleSet -> RuleSet

axiom add_R_collapses is
 fa(s : Rule, S : RuleSet)
  add_R(s, add_R(s, S)) = add_R(s, S)

op empty_R : RuleSet
```

```
op single_R : Rule -> RuleSet

op duo_R : Rule * Rule -> RuleSet

axiom definition_of_single_R is
 fa(s : Rule)
  single_R(s) = add_R(s, empty_R)

axiom definition_of_duo_R is
 fa(s1 : Rule, s2 : Rule)
  duo_R(s1, s2) = add_R(s1, single_R(s2))

axiom add_R_collapses is
 fa(r : Rule, R : RuleSet)
  add_R(r, add_R(r, R)) = add_R(r, R)

axiom false_is_refuted is
 fa(S : SentenceSet, R : RuleSet)
  refute(add_S(false_sentence, S), R)

axiom apply_rule_to_refute is
 fa (S : SentenceSet, s : Sentence, R : RuleSet, r : Rule)
  apply_rS(r, S, s) &
  refute(add_S(s, S), add_R(r, R)) =>
  refute(S, add_R(r, R))

op binaryResolution : Rule

op Clause? : Sentence -> Boolean

sort Clause  = {s : Sentence | Clause? s}

op Literal? : Clause -> Boolean

sort Literal = {c : Clause | Literal? c}

op disjunction : SentenceSet -> Sentence

axiom disjunction_of_true is
 fa (S : SentenceSet)
  disjunction(add_S(true_sentence, S)) = true_sentence

axiom disjunction_of_false is
 fa(S : SentenceSet)
  disjunction(add_S(false_sentence, S)) = disjunction(S)

conjecture disjunction_of_single_lemma is
 fa(s : Sentence)
  disjunction(single_S(s)) = s

conjecture disjunction_of_duo_false_lemma is
 fa(s : Sentence)
  disjunction(duo_S(s, false_sentence)) = s

axiom definition_of_binaryResolution is
 fa(P : Literal, C : Clause, D : Clause)
  apply_rS(binaryResolution,
```

```
           duo_S(disjunction(duo_S(neg(P), C)),
                 disjunction(duo_S(P, D))),
           disjunction(duo_S(C, D)))

conjecture binaryResolution_test is
  fa(P : Literal)
   apply_rS(binaryResolution, duo_S(neg(P), P), false_sentence)

conjecture applyBinaryResolution_test is
  fa(P : Literal)
   apply_rS(binaryResolution, duo_S(neg(P), P), false_sentence)

conjecture refutation_test is
  fa(P : Literal)
   refute(duo_S(neg(P), P
```

Appendix 3: Procedural Resolution Theorem Prover

Here is the procedural version of the propositional resolution theorem prover—ultimately it will be developed as a refinement of the declarative version.

```
Closure = spec
import /Library/Structures/Data/Sets/Polymorphic
op subset?: [a] Set a -> Set a -> Boolean
op  closure: [a] (Set a -> Set a) -> Set a -> Set a
%% Should have subsort predicate that subset? s (f s)
def closure f s =
  let newS = f s in
  if subset? newS s then s
    else closure f newS

%% Incremental version
op  closure1: [a] (Set a -> Set a) -> Set a -> Set a
def closure1 f s =
  closure (fn s -> union s (f s)) s

endspec


WTP = spec
import Closure
import /Library/Structures/Data/Sets/Polymorphic/AsLists

sort Sentence = | True
                | False
                | Var String
                | Neg Sentence
                | Conjunction SentenceSet
                | Disjunction SentenceSet

sort SentenceSet = Set Sentence

op  Clause? : Sentence -> Boolean
def Clause? s =
  case s of
    | Disjunction _ -> true
    | True -> true
    | False -> true
    | _ -> false

sort Clause  = {s : Sentence | Clause? s}
sort ClauseSet = Set Clause

op  Literal? : Clause -> Boolean
def Literal? s =
  case s of
    | True -> true
    | False -> true
    | Var _ -> true
```

```
    | _ -> false

sort Literal = {c : Clause | Literal? c}

op  makeConjunction: SentenceSet -> Sentence
def makeConjunction ss =
  if empty? ss
    then True
  else if member? ss False
        then False
       else Conjunction(delete ss True)

op  makeDisjunction: SentenceSet -> Sentence
def makeDisjunction ss =
  let ss = simplifyDisSS ss in
  if empty? ss
    then False
  else if member? ss True
        then True
       else Disjunction ss

sort Rule = Clause * Clause -> ClauseSet

sort RuleSet = Set Rule

op  binaryResolution : Rule
def binaryResolution(c1,c2) =
  case (c1,c2) of
    | (Disjunction s1,Disjunction s2) ->
      union (binaryResolution1 s1 s2)  (binaryResolution1 s2 s1)
    | _ -> empty

op  binaryResolution1: SentenceSet -> SentenceSet -> ClauseSet
def binaryResolution1 s1 s2 =
  fold (fn r -> fn e1 ->
        fold (fn r -> fn e2 ->
              case e2 of
                | Neg ne2 ->
                  if e1 = ne2
                    then insert r (makeDisjunction(union (delete s1 e1)
(delete s2 e2)))
                    else r
                | _ -> r)
          r s2)
    empty s1

op  simplifyDisSS: SentenceSet -> SentenceSet
def simplifyDisSS ss =
  fold (fn rss -> fn s ->
        let s = simplifyS s in
        case s of
          | False -> rss
          | Neg ns ->
            if member? rss ns
              then singleton(True)
              else insert rss s
          | _ ->
```

```
            let ns = Neg s in
            if member? rss ns
              then singleton(True)
              else insert rss s)
  empty ss

op  simplifyS: Sentence -> Sentence
def simplifyS s =
  case s of
    | Neg ns ->
      (case simplifyS ns of
        | Neg nns -> nns
        | ns1 -> Neg ns1)
    | Conjunction ss -> makeConjunction(ss)
    | Disjunction ss -> makeDisjunction(simplifyDisSS ss)
    | _ -> s

op  applyRuleSet: RuleSet -> ClauseSet -> ClauseSet
def applyRuleSet rs cs =
  fold (fn ncs -> fn rl ->
        fold (fn ncs -> fn cl1 ->
              fold (fn ncs -> fn cl2 ->
                    union ncs (rl(cl1,cl2)))
                ncs cs)
          ncs cs)
    cs rs

op  refute : ClauseSet * RuleSet -> Boolean
def refute(cs,rs) =
  member? (closure1 (applyRuleSet rs) cs) False

endspec

Examples = spec

import WTP

op  listToSet: [a] List a -> Set a
def listToSet l =
  foldl (fn (e,s) -> insert s e) empty l

def binaryResolution_test =
  let P = Var "p" in
  binaryResolution(makeDisjunction(singleton(Neg(P))),
makeDisjunction(singleton P)) = singleton(False)

def binaryResolution_test2 =
  let Q = Var "q" in
  let R = Var "r" in
  let S = Var "s" in
  binaryResolution(makeDisjunction(listToSet[Neg S,Neg R]),
                   makeDisjunction(listToSet[S,Q,R]))

def refute_test1 =
  let P = Var "p" in
  refute(listToSet
[makeDisjunction(singleton(Neg(P))),makeDisjunction(singleton P)],
```

```
        singleton(binaryResolution))

def refute_test2 =
  let P = Var "p" in
  let Q = Var "q" in
  let R = Var "r" in
  refute(listToSet[makeDisjunction(listToSet[Neg P,Q]),
                   makeDisjunction(listToSet[Neg P,R]),
                   makeDisjunction(singleton P),
                   makeDisjunction(listToSet[Neg Q,Neg R])],
        singleton(binaryResolution))

def refute_test3 =
  let P = Var "p" in
  let Q = Var "q" in
  let R = Var "r" in
  refute(listToSet[makeDisjunction(listToSet[Neg P,Q]),
                   makeDisjunction(listToSet[Neg P,R]),
                   makeDisjunction(singleton P),
                   makeDisjunction(listToSet[Neg Q,R])],
        singleton(binaryResolution))

def refute_test4 =
  let P = Var "p" in
  let Q = Var "q" in
  let R = Var "r" in
  let S = Var "s" in
  refute(listToSet[makeDisjunction(listToSet[Neg P,Neg R]),
                   makeDisjunction(listToSet[Neg Q,Neg R]),
                   makeDisjunction(singleton (P)),
                   makeDisjunction(listToSet[Neg S,R]),
                   makeDisjunction(listToSet[S,Q,R]),
                   makeDisjunction(singleton(Neg Q))],
        singleton(binaryResolution))



endspec), add_R(binaryResolution, empty_R))

endspec
```

Appendix 4: Declarative Specification of a Simple Meeting-Scheduling Problem

Our first experiments had to do with the impact of ordering advice on the performance of the theorem prover. Our first problem simply tested the ability of the theorem prover to solve scheduling problems without the benefit of special advice.

```
meeting_spec = spec

(* comment:
Art, Bob and Carol are free at all three available times
except for TimeA, TimeB, and TimeC, respectively.  Each
pair of them needs to have a meeting.
   end comment *)


  sort Time

  op Art : Time -> Boolean
  op Bob : Time -> Boolean
  op Carol : Time -> Boolean

  op TimeA : Time
  op TimeB : Time
  op TimeC : Time

  axiom Art_Free_Time is
   Art(TimeB) & Art(TimeC)

  axiom Bob_Free_Time is
   Bob(TimeA) & Bob(TimeC)

  axiom Carol_Free_Time is
   Carol(TimeA) & Carol(TimeB)

  conjecture meetinga is
   ex(time : Time)
    Bob(time) & Carol(time)

  conjecture meetingb is
   ex(time : Time)
    Art(time) & Carol(time)

  conjecture meetingc is
   ex(time : Time)
```

```
    Art(time) & Bob(time)

endspec
```

There are three people, Art, Bob, and Carol, who are busy at Time A, Time B, and Time C, respectively. We pose three independent scheduling problems in which each pair of them is to meet. These are simple problems, which the theorem prover was able to solve almost instantaneously (about .2 or .3 seconds, of which most of the time was spent processing assertions added by the interface for numerical reasoning which were extraneous to this problem).

Appendix 5: Procedural Meeting-Scheduling Theory

Ideally, this meeting scheduler would be obtained by specializing a procedural specification of a theorem prover to a declarative theory of meeting scheduling. Steps towards a declarative meeting-scheduling theory are outlined later in this section.

```
procedural_schedule = spec


 sort Person
 sort Time

 sort Appointment = Person * Time

 op noAppointment : Appointment

 sort Schedule = {a : Appointment}

 op noSchedule : Schedule

 op addAppointment : Person * Time * Schedule -> Schedule
 op removeAppointment : Person * Time * Schedule ->
Schedule

 sort Group = {p : Person}

 op emptyGroup? : Group -> Boolean

 op arbGroup : Group -> Person

 op restGroup : Group -> Group

 sort Org = {g : Group}

 op emptyOrg? : Org -> Boolean

 op arbOrg : Org -> Group

 op restOrg : Org -> Org

(* comment:
scheduleOrg schedules meetings for a class of groups so
that all persons in each group will have a meeting
together, with no others.  It accepts two parameters: a
```

```
schedule of already agreed-on meetings,  and a schedule of
people's free times.
   end-comment *)

 op scheduleOrg : Org * Schedule * Schedule -> Schedule
 op scheduleOneGroup : Group * Org * Schedule * Schedule ->
Schedule

(* comment:
if the organization, i.e., the class of groups is empty,
the meetin problem is already solved.  Othewise. we select
one group arbitrarily to schedule a meeting for.
   end-comment *)

 def scheduleOrg(C : Org,
           oldSchedule : Schedule,
           freeSchedule : Schedule): Schedule =
     if emptyOrg?(C)
     then oldSchedule
     else scheduleOneGroup(arbOrg(C),restOrg(C),
oldSchedule, freeSchedule)

       op scheduleOnePerson :
       Person * Group * Org * Schedule * Schedule ->
       Schedule

(* comment:
To schedule meetings for a group, select an arbitrary
person from the group; schedule a meeting for that person;
then schedule meetings for the others.
   end comment *)

 def scheduleOneGroup (group1 : Group,
                otherGroups : Org,
                oldSchedule : Schedule,
                freeSchedule : Schedule): Schedule =
     if emptyGroup?(group1)
     then scheduleOrg(otherGroups, oldSchedule,
freeSchedule)
     else scheduleOnePerson(arbGroup(group1),
                  restGroup(group1),
                  otherGroups,
                  oldSchedule,
                  freeSchedule)

(* comment:
```

```
getAppointment finds an arbitrary appointment (or free
time) in a person's schedule, if any
   end comment *)

 op getAppointment : Person * Schedule -> Appointment

(* comment:
removeAppointment removes that appointment from the
schedule, if one was found; otherwise, it leaves the
schedule unchanged.

inSchedule checks to see if a person has an appointment (or
free time) at a given time.
end comment *)

 op inSchedule : Person * Time * Schedule -> Boolean

% time yields the time of a given appointment

 op time : Appointment -> Time

(* comment
scheduleOnePerson schedules a meeting for a single person.
It finds a free time for that person; if none exists, it
fails.  It then attempts to schedule the rest of the group
to meet at that time.

If that fails, it attempts to reschedule a meeting for the
given person at some time other than the selected free
time. For this purpose,it removes that time from the
person's free time schedule.
   end comment *)

 def scheduleOnePerson
  (person1 : Person,
   restofGroup : Group,
   otherGroups : Org,
   oldSchedule : Schedule,
   freeSchedule : Schedule) : Schedule =
   let appointment = getAppointment(person1, freeSchedule)
in
    if ~(appointment = noAppointment)
    then let time1 = time(appointment) in
         let newFreeSchedule =
          removeAppointment(person1, time1, freeSchedule)
in
         let newSchedule = scheduleGroupTime(
```

```
           time1,
           restofGroup,
           otherGroups,
           addAppointment(person1, time1, oldSchedule),
           newFreeSchedule) in
      if ~(newSchedule = noSchedule)
      then scheduleOrg(otherGroups, newSchedule,
newFreeSchedule)
         else scheduleOnePerson
              (person1,
               restofGroup,
               otherGroups,
          oldSchedule,
             newFreeSchedule)

    else noSchedule

 op scheduleGroupTime :
       Time * Group * Org * Schedule * Schedule -> Schedule

(* comment:
scheduleGroupTime attempts to find a schedule in which an
entire group will meet at a given time.  If the group is
empty, it goes on to schedule the other groups.  Otherwise,
it selects an arbitrary person from that group, and checks
to see if that person is free at the given time.  If not,
it fails.  Otherwise, it continues by scheduling the rest
of the group at the given time.
   end comment *)

 def scheduleGroupTime(time1 : Time,
                       group1 : Group,
                       otherGroups : Org,
                       oldSchedule : Schedule,
                  freeSchedule : Schedule) : Schedule =

 if emptyGroup?(group1)
 then scheduleOrg(otherGroups, oldSchedule, freeSchedule)
 else let person1 = arbGroup(group1) in
      if inSchedule(person1, time1, freeSchedule)
      then scheduleGroupTime
     (time1,
      restGroup(group1),
      otherGroups,
      addAppointment(person1, time1, oldSchedule),
      removeAppointment(person1, time1, freeSchedule))
      else noSchedule
```

```
endspec
```

Appendix 6: Two-Meeting Scheduling Problem

In the next problem we tried to introduce an asymmetry by having one of the participants, Bob, have more free time than the other two. We pose a single problem in which two distinct meetings are to be scheduled, between Bob and his two colleagues.

```
two_meeting_spec = spec

(* comment:
Art is free only at TimeA; Carol is free only at TimeB; Bob
is free both times.  They need to have two meetings, one
between Art and Bob, the other between Bob and Carol.
   end comment *)

  sort Time

  op Art : Time -> Boolean
  op Bob : Time -> Boolean
  op Carol : Time -> Boolean

  op TimeA : Time
  op TimeB : Time

  axiom Times_Distinct is
  ~(TimeA = TimeB)

  axiom Art_Free_Time is
   Art(TimeA)

  axiom Bob_Free_Time is
   Bob(TimeA) & Bob(TimeB)

  axiom Carol_Free_Time is
   Carol(TimeB)


  conjecture two_meetings is
   ex(time1 : Time, time2 : Time)
    Art(time1) & Bob(time1) & Bob(time2) & Carol(time2) &
~(time1 = time2)


endspec
```

Again this problem was too simple to pose much difficulty to the theorem prover; it was solved almost instantaneously.

Appendix 7: Scheduling with Space

We next introduced the element of scheduling space as well as time. We assume there are two rooms, one of which is free only in the morning, the other in the afternoon. The scheduler is to find both times and rooms for two meetings.

```
meeting_room_spec = spec

(* comment:
Art is free only in the morning; Carol is free only in the
afternoon; Bob is free both times. There are two meeting
rooms.  The light room is free only in the morning; the
dark room is free only in the afternoon.  We need to
schedule two meetings, one for Art and Bob and another for
Bob and Carol.  The schedule is to provide room
assignments.
   end comment *)

  sort Person
  sort Room
  sort Time

  op Art : Person
  op Bob : Person
  op Carol : Person

  op DarkRoom : Room
  op LightRoom : Room

  op Morning : Time
  op Afternoon : Time

  op FreePerson : Person * Time -> Boolean
  op FreeRoom : Room * Time -> Boolean

  axiom Times_Distinct is
  ~(Morning = Afternoon)

  axiom Rooms_Distinct is
  ~(DarkRoom = LightRoom)

  axiom Art_Free_Time is
   FreePerson(Art, Morning)

  axiom Bob_Free_Time is
```

```
   FreePerson(Bob, Morning) & FreePerson(Bob, Afternoon)

  axiom Carol_Free_Time is


  axiom LightRoom_Free_Time is
   FreeRoom(LightRoom, Morning)

  axiom DarkRoom_Free_Time is
   FreeRoom(DarkRoom, Afternoon)


  conjecture two_meeting_rooms is
   ex(time1 : Time, time2 : Time, room1 : Room, room2 :
Room,
      schedule : Time * Room * Time * Room)
    FreePerson(Art, time1) & FreePerson(Bob, time1) &
    FreeRoom(room1, time1) &
    FreePerson(Bob, time2) & FreePerson(Carol, time2) &
    FreeRoom(room2, time2) &
    ~(time1 = time2) &
    schedule = (time1, room1, time2, room2)
endspec
```

Appendix 8: Advice about Constraint Ordering

All the above problems were so simple that they were not useful in experimenting with the effects of advice. In our next problem we attempted to require a meeting to be scheduled for many participants and rooms.

```
ordered_hard_meeting_room_spec = spec

(* comment:
There are nine people, eight times, and eleven rooms.  All
people and rooms are available at all times, except Bob and
the dark room are available only at 5.  It is necessary to
schedule a time and room for a single meeting with
everybody.  It was hypothesized that an ordering that
scheduled Bob and the dark room first would dominate other
strategies.
  end comment *)

  sort Person
  sort Room
  sort Time

  op Art : Person
  op Bob : Person
  op Carol : Person
  op Don : Person
  op Ed : Person
  op Frank : Person
  op George : Person
  op Harry : Person
  op Ivy : Person


  op DarkRoom : Room
  op LightRoom : Room
  op RedRoom : Room
  op OrangeRoom : Room
  op YellowRoom : Room
  op GreenRoom : Room
  op BlueRoom : Room
  op IndigoRoom : Room
  op VioletRoom : Room
  op BlackRoom : Room
  op WhiteRoom : Room
```

```
op Time1 : Time
op Time2 : Time
op Time3 : Time
op Time4 : Time
op Time5 : Time
op Time6 : Time
op Time7 : Time
op Time8 : Time

op FreePerson : Person * Time -> Boolean
op FreeRoom : Room * Time -> Boolean

axiom Art_Free_Time is
  FreePerson(Art, Time1) &
  FreePerson(Art, Time2) &
  FreePerson(Art, Time3) &
  FreePerson(Art, Time4) &
  FreePerson(Art, Time5) &
  FreePerson(Art, Time6) &
  FreePerson(Art, Time7) &
  FreePerson(Art, Time8)

axiom Bob_Free_Time is
 FreePerson(Bob, Time5)

axiom Carol_Free_Time is
  FreePerson(Carol, Time1) &
  FreePerson(Carol, Time2) &
  FreePerson(Carol, Time3) &
  FreePerson(Carol, Time4) &
  FreePerson(Carol, Time5) &
  FreePerson(Carol, Time6) &
  FreePerson(Carol, Time7) &
  FreePerson(Carol, Time8)

axiom Don_Free_Time is
  FreePerson(Don, Time1) &
  FreePerson(Don, Time2) &
  FreePerson(Don, Time3) &
  FreePerson(Don, Time4) &
  FreePerson(Don, Time5) &
  FreePerson(Don, Time6) &
  FreePerson(Don, Time7) &
  FreePerson(Don, Time8)


axiom Ed_Free_Time is
```

```
  FreePerson(Ed, Time1) &
  FreePerson(Ed, Time2) &
  FreePerson(Ed, Time3) &
  FreePerson(Ed, Time4) &
  FreePerson(Ed, Time5) &
  FreePerson(Ed, Time6) &
  FreePerson(Ed, Time7) &
  FreePerson(Ed, Time8)


axiom Frank_Free_Time is
  FreePerson(Frank, Time1) &
  FreePerson(Frank, Time2) &
  FreePerson(Frank, Time3) &
  FreePerson(Frank, Time4) &
  FreePerson(Frank, Time5) &
  FreePerson(Frank, Time6) &
  FreePerson(Frank, Time7) &
  FreePerson(Frank, Time8)


axiom George_Free_Time is
  FreePerson(George, Time1) &
  FreePerson(George, Time2) &
  FreePerson(George, Time3) &
  FreePerson(George, Time4) &
  FreePerson(George, Time5) &
  FreePerson(George, Time6) &
  FreePerson(George, Time7) &
  FreePerson(George, Time8)


axiom Harry_Free_Time is
  FreePerson(Harry, Time1) &
  FreePerson(Harry, Time2) &
  FreePerson(Harry, Time3) &
  FreePerson(Harry, Time4) &
  FreePerson(Harry, Time5) &
  FreePerson(Harry, Time6) &
  FreePerson(Harry, Time7) &
  FreePerson(Harry, Time8)

axiom Ivy_Free_Time is
  FreePerson(Ivy, Time1) &
  FreePerson(Ivy, Time2) &
  FreePerson(Ivy, Time3) &
  FreePerson(Ivy, Time4) &
```

```
    FreePerson(Ivy, Time5) &
    FreePerson(Ivy, Time6) &
    FreePerson(Ivy, Time7) &
    FreePerson(Ivy, Time8)

  axiom LightRoom_Free_Time is
   FreeRoom(LightRoom, Time1) &
   FreeRoom(LightRoom, Time2) &
   FreeRoom(LightRoom, Time3) &
   FreeRoom(LightRoom, Time4) &
%   FreeRoom(LightRoom, Time5) &
   FreeRoom(LightRoom, Time6) &
   FreeRoom(LightRoom, Time7) &
   FreeRoom(LightRoom, Time8)

  axiom DarkRoom_Free_Time is
%   FreeRoom(DarkRoom, Time1) &
%   FreeRoom(DarkRoom, Time2) &
%   FreeRoom(DarkRoom, Time3) &
%   FreeRoom(DarkRoom, Time4) &
   FreeRoom(DarkRoom, Time5)% &
%   FreeRoom(DarkRoom, Time6) &
%   FreeRoom(DarkRoom, Time7) &
%   FreeRoom(DarkRoom, Time8)


  axiom RedRoom_Free_Time is
   FreeRoom(RedRoom, Time1) &
   FreeRoom(RedRoom, Time2) &
   FreeRoom(RedRoom, Time3) &
   FreeRoom(RedRoom, Time4) &
%   FreeRoom(RedRoom, Time5) &
   FreeRoom(RedRoom, Time6) &
   FreeRoom(RedRoom, Time7) &
   FreeRoom(RedRoom, Time8)


  axiom OrangeRoom_Free_Time is
   FreeRoom(OrangeRoom, Time1) &
   FreeRoom(OrangeRoom, Time2) &
   FreeRoom(OrangeRoom, Time3) &
   FreeRoom(OrangeRoom, Time4) &
%   FreeRoom(OrangeRoom, Time5) &
   FreeRoom(OrangeRoom, Time6) &
   FreeRoom(OrangeRoom, Time7) &
   FreeRoom(OrangeRoom, Time8)
```

```
  axiom YellowRoom_Free_Time is
   FreeRoom(YellowRoom, Time1) &
   FreeRoom(YellowRoom, Time2) &
   FreeRoom(YellowRoom, Time3) &
   FreeRoom(YellowRoom, Time4) &
%   FreeRoom(YellowRoom, Time5) &
   FreeRoom(YellowRoom, Time6) &
   FreeRoom(YellowRoom, Time7) &
   FreeRoom(YellowRoom, Time8)

  axiom GreenRoom_Free_Time is
   FreeRoom(GreenRoom, Time1) &
   FreeRoom(GreenRoom, Time2) &
   FreeRoom(GreenRoom, Time3) &
   FreeRoom(GreenRoom, Time4) &
%   FreeRoom(GreenRoom, Time5) &
   FreeRoom(GreenRoom, Time6) &
   FreeRoom(GreenRoom, Time7) &
   FreeRoom(GreenRoom, Time8)

  axiom BlueRoom_Free_Time is
   FreeRoom(BlueRoom, Time1) &
   FreeRoom(BlueRoom, Time2) &
   FreeRoom(BlueRoom, Time3) &
   FreeRoom(BlueRoom, Time4) &
%   FreeRoom(BlueRoom, Time5) &
   FreeRoom(BlueRoom, Time6) &
   FreeRoom(BlueRoom, Time7) &
   FreeRoom(BlueRoom, Time8)

  axiom IndigoRoom_Free_Time is
   FreeRoom(IndigoRoom, Time1) &
   FreeRoom(IndigoRoom, Time2) &
   FreeRoom(IndigoRoom, Time3) &
   FreeRoom(IndigoRoom, Time4) &
%   FreeRoom(IndigoRoom, Time5) &
   FreeRoom(IndigoRoom, Time6) &
   FreeRoom(IndigoRoom, Time7) &
   FreeRoom(IndigoRoom, Time8)

  axiom VioletRoom_Free_Time is
   FreeRoom(VioletRoom, Time1) &
   FreeRoom(VioletRoom, Time2) &
   FreeRoom(VioletRoom, Time3) &
   FreeRoom(VioletRoom, Time4) &
%   FreeRoom(VioletRoom, Time5) &
   FreeRoom(VioletRoom, Time6) &
```

```
   FreeRoom(VioletRoom, Time7) &
   FreeRoom(VioletRoom, Time8)

  conjecture ordered_hard_meeting_room is
   ex(time : Time, room : Room,
      schedule : Time * Room)
    FreePerson(Art, time) &
    FreePerson(Bob, time) &
    FreePerson(Carol, time) &
    FreePerson(Don, time) &
    FreePerson(Ed, time) &
    FreePerson(Frank, time) &
    FreePerson(George, time) &
    FreePerson(Harry, time) &
    FreePerson(Ivy, time) &
    FreeRoom(room, time) &
    schedule = (time, room)


  def meeting_prove_options =
 "
 (use-resolution t)
 (use-hyperresolution nil)
 (use-negative-hyperresolution nil)
 (use-paramodulation)
 (use-factoring)
 (use-literal-ordering-with-hyperresolution 'literal-
ordering-p)
 (use-literal-ordering-with-negative-hyperresolution
'literal-ordering-p)
 (use-literal-ordering-with-resolution 'literal-ordering-a)
 (use-literal-ordering-with-paramodulation 'literal-
ordering-p)
 (use-ac-connectives)
 (run-time-limit 10)
 (assert-supported nil)
 (use-code-for-numbers nil)
 (print-symbol-ordering)
 (print-final-rows)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Art|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Carol|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Don|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Ed|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Frank|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|George|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Harry|)
 (declare-ordering-greaterp 'snark::|Bob| 'snark::|Ivy|)
```

```
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|LightRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|RedRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|OrangeRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|YellowRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|GreenRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|BlueRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|IndigoRoom|)
 (declare-ordering-greaterp 'snark::|DarkRoom|
'snark::|VioletRoom|)
 "


endspec
```

Note that for this problem Bob is the busiest person—he is free only at Time 5—and the dark room is the busiest room—it is also free only at 5. The other people are available at all times and the other rooms are available at all times other than Time 5.

Our expectation was that if we gave the theorem prover advice to schedule the most constrained resources—Bob and the dark room—first, it would exhibit better performance. We did separate runs in which the advice was absent or perversely bad—we actually told it to schedule the most constrained resources last instead of first:

```
 (declare-ordering-greaterp 'snark::|Art|  'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|Carol| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|Don| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|Ed| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|Frank| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|George| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|Harry| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|Ivy| 'snark::|Bob|)
 (declare-ordering-greaterp 'snark::|LightRoom|
'snark::|DarkRoom|)
 (declare-ordering-greaterp 'snark::|RedRoom|
'snark::|DarkRoom|)
 (declare-ordering-greaterp 'snark::|OrangeRoom|
'snark::|DarkRoom|)
```

```
 (declare-ordering-greaterp 'snark::|YellowRoom|
'snark::|DarkRoom|)
 (declare-ordering-greaterp 'snark::|GreenRoom|
'snark::|DarkRoom|)
 (declare-ordering-greaterp 'snark::|BlueRoom|
'snark::|DarkRoom|)
 (declare-ordering-greaterp 'snark::|IndigoRoom|
'snark::|DarkRoom|)
 (declare-ordering-greaterp 'snark::|VioletRoom|
'snark::|DarkRoom|)
```

Appendix 9: Preference Advice

Up to now, the advice we examined involved help in finding a solution. But there is another kind of advice that involves expressing a preference for one solution over another. This is different from a constraint, because we are assuming that either solution is acceptable; but if both are possible, one of them is preferable to the other.

We experimented with one way of handling this sort of advice. We first ignore the preference advice and find one solution to the problem. We then seek another solution to the same problem, adding to the specification the condition that the new solution be better than the old. Here "better" means preferable in the sense specified by the advice. If a better solution is found, we then seek a third solution, where the third solution is constrained to be better than the second, and so on until no more solutions exist or a time limit is exceed. In this way, we continue to find better and better solutions until the search space or time limit is exhausted.

```
preference_meeting_spec = spec


  sort Person
  sort Room
  sort Time = Integer

  op Alice : Person
  op Bob : Person
  op Carol : Person
  op Don : Person
  op Ellen : Person
  op Frank : Person

  op FreePerson : Person * Time -> Boolean

  axiom Alice_Free_Time is
    FreePerson(Alice, 1) &
    FreePerson(Alice, 2) &
    FreePerson(Alice, 3) &
    FreePerson(Alice, 4)

  axiom Bob_Free_Time is
    FreePerson(Bob, 1) &
    FreePerson(Bob, 2) &
    FreePerson(Bob, 3) &
    FreePerson(Bob, 4)
```

```
   axiom Carol_Free_Time is
    FreePerson(Carol, 1) &
    FreePerson(Carol, 2) &
    FreePerson(Carol, 3) &
    FreePerson(Carol, 4)

  axiom Don_Free_Time is
    FreePerson(Don, 1) &
    FreePerson(Don, 2) &
    FreePerson(Don, 3) &
    FreePerson(Don, 4)

  axiom Ellen_Free_Time is
    FreePerson(Ellen, 1) &
    FreePerson(Ellen, 2) &
    FreePerson(Ellen, 3) &
    FreePerson(Ellen, 4)

  axiom Frank_Free_Time is
    FreePerson(Frank, 1) &
    FreePerson(Frank, 2) &
    FreePerson(Frank, 3) &
    FreePerson(Frank, 4)


  conjecture preference_meeting is
   ex(time1 : Time, time2 : Time, time3 : Time, time4 :
Time,
      rating : Nat,
      schedule : Nat * Time * Time * Time * Time)
    FreePerson(Alice, time1) & FreePerson(Bob, time1) &
    FreePerson(Carol, time2) & FreePerson(Don, time2) &
    FreePerson(Ellen, time3) & FreePerson(Frank, time3) &

    FreePerson(Alice, time4) & FreePerson(Bob, time4) &
    FreePerson(Carol, time4) & FreePerson(Don, time4) &
    FreePerson(Ellen, time4) & FreePerson(Frank, time4) &
    ~(time4 =  time1) & ~(time4 = time2) & ~(time4 =
time3) &
    rating = max(time1, max(time2, max(time3, time4))) &
    schedule = (rating, time1, time2, time3, time4)
endspec
```

In this problem there are three afternoon meetings to be scheduled between Alice and Bob, Carol and Don, and Ellen and Frank, respectively, and a fourth meeting to be scheduled between all of them together. Although the meetings could all be scheduled at different times, we specify a preference that we would like the meetings to be over as

early as possible. While some schedules have the last meeting over at 4, with this advice the theorem prover finds several acceptable plans but ultimately zeroes in on one in which the three smaller meetings are all at the same time, 1, and the fourth meeting is scheduled immediately after, with an optimal finishing time of 2.

Appendix 10: Opportunistic Meeting Scheduling

We have given advice about preferences with respect to time; we can also express preferences with respect to distance. Also this is the first example in which we deal with concrete dates rather than abstract times. This problem is expressed in the language of SNARK, which is more primitive than Specware's language, because we are using SNARK temporal-reasoning features that are not yet made available through the Specware interface.

```
(defun OPPORTUNISTIC-MEETING-PLAN ()
  (new-row-context)
  (declare-constant 'alice :sort 'person)
  (declare-constant 'bob :sort 'person)

  (assert '(travel-from-to
         alice
         (feature populated-place San-Francisco
               (feature 1st-order-division-countries
California United-States))
         (feature populated-place Minneapolis
               (feature 1st-order-division-countries
Minnesota United-States))
         (date-interval 2005 3 31 12 :until 2005 3 31 16)))

  (assert '(in
         alice
         (feature populated-place Minneapolis
               (feature 1st-order-division-countries
Minnesota United-States))
         (date-interval 2005 4 1 :until 2005 4 5)))

  (assert '(travel-from-to
         alice
         (feature populated-place Minneapolis
               (feature 1st-order-division-countries
Minnesota United-States))
         (feature populated-place San-Francisco
               (feature 1st-order-division-countries
California United-States))
         (date-interval 2005 4 6 5 :until 2005 4 6 7)))

  (assert '(travel-from-to
         bob
         (feature populated-place New-York
```

```
               (feature 1st-order-division-countries New-
York United-States))
         (feature populated-place Saint-Paul
               (feature 1st-order-division-countries
Minnesota United-States))
         (date-interval 2005 4 2 3 :until 2005 4 2 6)))

  (assert '(in bob
            (feature populated-place Saint-Paul (feature
1st-order-division-countries Minnesota United-States))
            (date-interval 2005 4 3 :until 2005 4 7)))

  (assert '(travel-from-to
         bob
         (feature populated-place Saint-Paul
               (feature 1st-order-division-countries
Minnesota United-States))
         (feature populated-place New-York
               (feature 1st-order-division-countries New-
York United-States))
         (date-interval 2005 4 8 9 :until 2005 4 8 15)))

  (prove '
    (could-meet-in-place alice bob (date-interval 2005 4 1
:until 2005 4 10) ?region1 ?region2)
         :answer '(Near ?region1 ?region2))

)
```

In this problem, Alice and Bob need to meet during the first ten days of April, but Alice lives in San Francisco and Bob lives in New York. We give as advice the preference to reduce travel distances. They could schedule a meeting in New York, San Francisco, or someplace in between, which would mean a long trip for one or both of them. However, examining the itineraries of Alice and Bob, the theorem prover sees that Alice has a trip to Minneapolis, Minnesota, on April 3–7, and Bob has a trip to Saint Paul, on April 5–9. Both these trips are within the desired time interval and their own time intervals overlap. Furthermore, using procedural attachments to a gazetteer and geographical computation software, the theorem prover is able to establish that Minneapolis and Saint Paul are only 6 miles apart, a much shorter distance than 3000 miles. Hence a meeting during the intersection of the time intervals, April 5–7, in either of the Twin Cities, will be preferable to a special trip.

Appendix 11: Learning in Meeting Scheduling

We next looked at a certain kind of learning in the context of meeting scheduling. We imagine an impossible meeting-scheduling problem, in which all the participants are so constrained that there is no possible solution:

```
impossible_meeting_spec = spec

(* comment:

Here is an example of an unsolvable meeting problem: Art,
Bob, and Carol are to have separate meetings between each
pair of them, but they are only free at Time1 and Time2.
The proof search generates a number of two-literal clauses,
each of which presents a satisfactory schedule if only two
of them were also free at Time3.  In other words, abduction
could be used to suggest relaxations of contraints.
end comment *)

  sort Person
  sort Room
  sort Time

  op Art : Person
  op Bob : Person
  op Carol : Person

  op Time1 : Time
  op Time2 : Time
  op Time3 : Time

 op FreePerson : Person * Time -> Boolean

  axiom Times_Distinct is
    ~(Time1 = Time2) &
    ~(Time1 = Time3) &
    ~(Time2 = Time3)

  axiom Art_Free_Time is
    FreePerson(Art, Time1) &
    FreePerson(Art, Time2)

  axiom Bob_Free_Time is
   FreePerson(Bob, Time1) &
   FreePerson(Bob, Time2)
```

```
  axiom Carol_Free_Time is
    FreePerson(Carol, Time1) &
    FreePerson(Carol, Time2)

  conjecture impossible_meetings is
   ex(time1 : Time, time2 : Time, time3 : Time, schedule :
Time * Time * Time)
    FreePerson(Art, time1) & FreePerson(Bob, time1) &
    FreePerson(Bob, time2) & FreePerson(Carol, time2) &
~(time2 = time1) &
    FreePerson(Art, time3) & FreePerson(Carol, time3) &
~(time3 =  time1) & ~(time3 = time2) &
    schedule = (time1, time2, time3)

endspec
```

Here we need to schedule distinct meetings between Art and Bob, between Bob and Carol, and between Art and Carol, but all of them are free at only two times. If we examine the failed proof, we find that all branches of the search space end in two-unit clauses, in which we hope to achieve that two of the participants are free at the third time, Time 3. These clauses are the negated forms of

```
FreePerson(Art, Time3) & FreePerson(Bob, Time3),

FreePerson(Bob, Time3) & FreePerson(Carol, Time3),
```

etc. This suggests that if the constraints of the problem were relaxed so that two of the participants were free at Time 3, the problem would be solvable. Detecting such conditions is carried out by the abduction facility of SNARK (or other theorem provers).

Appendix 12: Type-Inference Theory

Many of the same techniques that we apply to meeting scheduling can also be applied to type inference, the coherent assignment of types to the symbols in an expression. The purpose of looking at type checking is to ensure that the techniques we develop for meeting scheduling do indeed have more general applicability. Also, Specware itself uses type inference so any improved implementations we obtain will be of immediate use.

Here is a procedural version of a type inference system.

```
AbstractSyntax =
spec
  type Expr = | Var String
              | IntConst Integer
              | RealConst(Integer * Nat)
              | Tuple(Expr * Expr)
             %| Tuple(List Expr)
              | Apply(Expr * Expr)
  type Type = | Integer
              | Real
              | Product(Type * Type)
             %| Product(List Type)
              | Arrow(Type * Type)
endspec

Typing =
spec
  import AbstractSyntax

  op TypeOf: Expr -> Type

  axiom typeOfIntConst is
    fa(i:Integer) TypeOf(IntConst i) = Integer

  axiom typeOfRealConst is
    fa(i:Integer,n:Nat) TypeOf(RealConst (i,n)) = Real

  axiom typeOfPlus is
    TypeOf(Var "+") =
Arrow(Product(Integer,Integer),Integer)
   or TypeOf(Var "+") = Arrow(Product(Real,Real),Real)

  axiom typeOfTuplePair is
    fa(e1: Expr,e2:Expr)
```

```
TypeOf(Tuple(e1,e2)) = Product(TypeOf e1,TypeOf e2)

  axiom typeOfApply is
    fa(e:Expr,ed:Expr,dom:Type,rng:Type)
    TypeOf e = Arrow(dom,rng) & TypeOf ed = dom
      => TypeOf(Apply(e,ed)) = rng

endspec

TypingExamples =
spec
  import Typing

  conjecture well_typed0 is
  ex(t:Type)
   TypeOf(IntConst 1) = t

  conjecture well_typed1 is
  ex(t:Type)
   TypeOf(Tuple(IntConst 1,IntConst 2)) = t

 conjecture well_typed2 is
  ex(t:Type)
   TypeOf(Apply(Var "+",Tuple(IntConst 1,IntConst 2))) = t

  conjecture badly_typed1 is
  ex(t:Type)
   TypeOf(Apply(Var "+",Tuple(IntConst 1,RealConst(2,1))))
= t

  conjecture badly_typed2 is
  ex(t:Type)
   TypeOf(Apply(Var "+",IntConst 1)) = t

endspec

Prove_well_typed0 =
  prove well_typed0 in TypingExamples
  answerVar t:Type

Prove_well_typed1 =
  prove well_typed1 in TypingExamples
  %answerVar t:Type

Prove_well_typed2 =
  prove well_typed2 in TypingExamples
  %answerVar t:Type
```

References

.[Apt] Apt, Krzysztof R. (2001). Theory and Practice of Logic Programming. Cambridge University Press, New York, NY

[Boyer] Boyer, R. S. and Moore, J S. (1981) Metafunctions: proving them correct and using them efficiently as new proof procedures.
In Boyer, R. S. and Moore, J S. (eds.), *The Correctness Problem in Computer Science*, pp. 103--184. Academic Press.

[Huet] Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Kestrel] Kestrel Institute (2004). Specware. http://www.specware.org/

[Meseguer] José Meseguer. General logics. In H.-D. Ebbinghaus, editor, *Logic Colloquium '87*, pages 275-329, Granada, Spain, July 1987. North-Holland.

[Ridge] Ridge, Tom (2004). A Mechanically Verified, Efficient, Sound and Complete Theorem Prover For First Order Logic. in Project: Archive of Formal Proofs**,** http://afp.sourceforge.net/entries/Verified-Prover.shtml

[Smith et al.] Marcel Becker, Limei Gilham, Douglas R. Smith, Planware II: Synthesis of Schedulers for Complex Resource Systems, 2003.

[Weyhrauch] Weyhrauch, Richard (1980). *Prolegomena to a Theory of Mechanized Formal Reasoning*. Artificial Intelligence, 13(1):133--176, 1980. 14**.**